Fall 11-30-2018

Controller Evolution and Divergence: A Software Perspective

Balaji Balasubramaniam
University of Nebraska - Lincoln, balaji@huskers.unl.edu

CONTROLLER EVOLUTION AND DIVERGENCE: A SOFTWARE PERSPECTIVE

by

Balaji Balasubramaniam

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Justin Bradley and Professor Sebastian Elbaum

Lincoln, Nebraska

30 November, 2018

CONTROLLER EVOLUTION AND DIVERGENCE: A SOFTWARE PERSPECTIVE

Balaji Balasubramaniam, M.S.

University of Nebraska, 2018

Advisers: Justin Bradley and Sebastian Elbaum

Successful controllers evolve as they are refined, extended, and adapted to new systems and contexts. This evolution occurs in the controller design and also in its software implementation. Model-based design and controller synthesis can help to synchronize this evolution of design and software, but such synchronization is rarely complete as software tends to also evolve in response to elements rarely present in a control model, leading to mismatches between the control design and the software.

In this thesis, we perform a first-of-its-kind study on the evolution of two popular open-source safety-critical autopilot control software – ArduPilot, and Paparazzi, to better understand how controllers evolve and the space of potential mismatches between control design and their software implementation. We then use that understanding to prototype a technique, called mutation tool, that can generate mutated versions of code to mimic evolution to assess its impact on a controller's behavior.

We report on three major findings. First, control software evolves quickly and controllers are rewritten in their entirety, many times over through the controller's lifetime, which implies that the design, synthesis, and implementation of controllers must support not just the initial baseline system but also their incremental evolution. Second, many software changes stem from an inherent mismatch between the continuous time/space physical model and its corresponding discrete software implementation, but also from the mishandling of exceptional conditions, and limitations and distinct data representation of the underlying computing architecture. Third, using our mutation tool that we developed,

we show that small code changes can have a dramatic effect in a controller's behavior, which implies that further support is needed to bridge these mismatches as carefully verified model properties may not necessarily translate to its software implementation.

DEDICATION

To my parents Balsubramaniam and Valarmathy. Thank you, you are my inspiration and my foundation. To my wife Ramya. I thank you for being so supportive.

ACKNOWLEDGMENTS

I would like to express my special appreciation and thanks to my advisors, Dr. Justin Bradley and Dr. Sebastian Elbaum, for their supervision and support towards the completion of this work. I would also like to thank Dr. Thanh Vu Nguyen for agreeing to be a part of my committee. I would like to acknowledge the valued assistance from the members of NIMBUS Lab.

A special thanks to my family for their constant support and encouragement. Finally, I would like to thank my friends for helping me get through the last two years.

Thank You!

GRANT INFORMATION

This work was partially supported by National Science Foundation under awards #1638099. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

Table of Contents

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Motivation . . . 2

1.2 Contributions . . . 3

1.3 Innovations . . . 6

2 Related work 7

2.1 Software development and control design . . . 7

2.2 Control software validation and verification . . . 8

2.2.1 Software evolution . . . 9

2.2.2 Mutation testing . . . 9

3 Controller evolution: an empirical study 11

3.1 Study . . . 11

3.1.1 Analysis Artifacts . . . 12

3.1.2 Analysis Process . . . 17

3.1.3 Threats to validity . . . 21

3.2 Results . . . 22

3.2.1 Answers to **RQ1** - How Much do Controllers Evolve? 22
3.2.2 Answers to **RQ2** - What Evolution Results from Model and Software Mismatches? 26
3.3 Conclusions 30

4 Impact of Software Changes on Control Performance **32**
4.1 Mutation testing process 32
4.2 Mutation tool overview 34
4.3 Methodolgy 36
4.3.1 Phase I 36
4.3.1.1 Template creation 37
4.3.2 Phase II 39
4.3.2.1 MATLAB specific mutation 40
4.3.3 Phase III 41
4.4 Results 42
4.4.1 Analysis I 43
4.4.2 Analysis II 45

5 Conclusions and Future Work **50**

Bibliography **53**

List of Figures

1.1 Evolution at the design and software implementation levels. 1

3.1 Overview of Study Analysis Process 11

4.1 Mutation testing process overview. 33
4.2 Mutation Tool Architecture. 35
4.3 Cruise control system divergence for speed step quantity 47
4.4 Helicopter system divergence for pitch angle step quantity 48
4.5 Helicopter system divergence for roll angle step quantity 48
4.6 Boeing747 system divergence for airspeed step quantity 49
4.7 Boeing747 system divergence for altitude step quantity 49

List of Tables

3.1 ArduPilot Files Examined 12
3.2 Paparazzi Files Examined 12
3.3 Definitions of Categories for Mismatches Between Models and Software 13
3.4 Examples of Categories for ArduPilot Mismatches Between Models and Software in ArduPilot 15
3.5 Examples of Categories for Paparazzi UAV Mismatches Between Models and Software in Paparazzi UAV 18
3.6 Overview of ArduPilot Control Software Evolution 23
3.7 Overview of Paparazzi UAV Control Software Evolution 24
3.8 Classification Results of ArduPilot Mismatches Between Models and Software 26
3.9 Classification Results of Paparazzi UAV Mismatches Between Models and Software 26

4.1 5 top most changes from Precision and Accuracy category grouped by similarity 38
4.2 5 top most changes from Exception Handling category grouped by similarity 38
4.3 5 top most changes from Time and Space Model category grouped by similarity 39
4.4 Mutation output details for Precision and Accuracy category in all three systems 41
4.5 Mutation output details for Exception handling category in all three systems . 42
4.6 Mutation output details for Time and Space Model category in all three systems 42

4.7 Summary of the mutation output details for all three systems 42
4.8 Summary of mutation results for all three systems 43

Chapter 1

Introduction

Successful systems evolve, and so do their controllers. Conceptually, this evolution occurs at two distinct levels as shown in Figure 1.1. At the control design level, that evolution may occur on the mathematical representations or higher level models in the chosen representation (e.g., Simulink, MATLAB, Octave). At this level it is common to observe model changes meant to refine the control law as the logical conditions under which a system should operate are realized, as the assumptions or levels of abstraction of the model are refined, or the model is revised to fit another system.

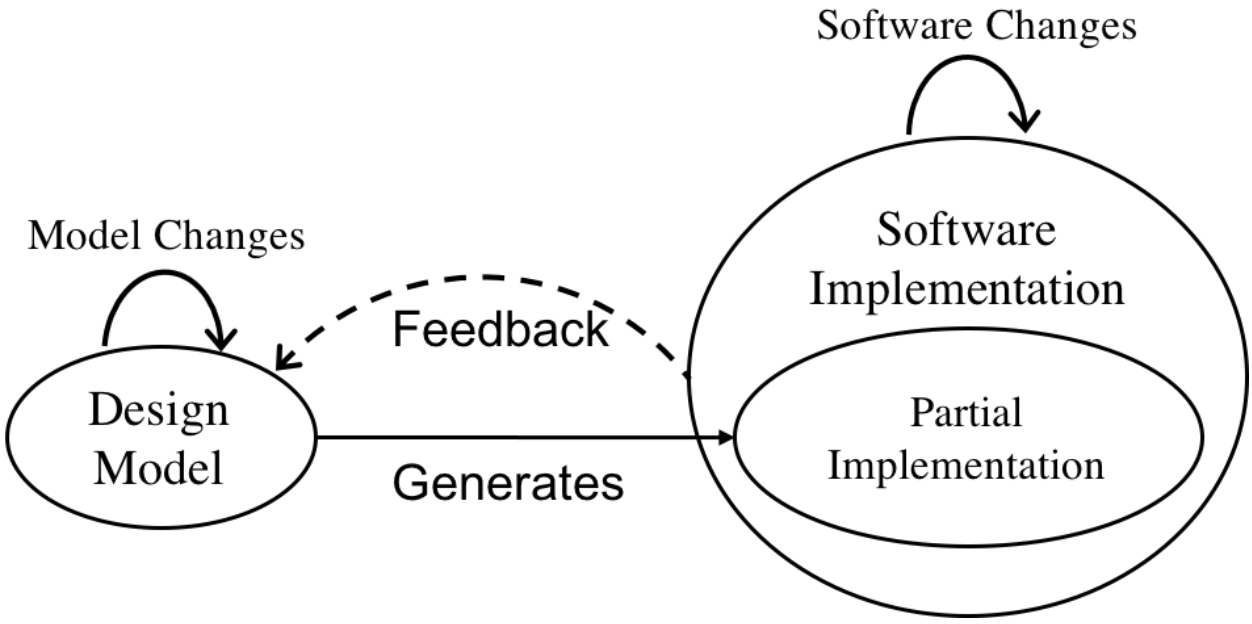


Figure 1.1: Evolution at the design and software implementation levels.

At the software level of the controller we observe at least three types of changes. First, software changes that directly map to the same changes in the control design. These changes constitute the primary target for tools supporting model-based design [61, 35] or controller synthesis [11]. Second, software changes that are meant to complete pieces of the implementation that were not defined in the design, either because of the level of modeling abstraction employed, or because it was not cost-effective to define them at the design level. Third, changes driven by the need to integrate the software with a larger software ecosystem that goes beyond the controller itself, or by software maintenance needs.

1.1 Motivation

In this section, we discuss the need for understanding the controller code evolution and the need to analyse the impact in safety-critical systems. The frequency of each type of software change varies significantly across systems. For a selected number of safety-critical software with large development resources, most changes can occur at the design level and be automatically verified and transfered to code with high fidelity (as shown by the arrow in Figure 1.1 going from the design model to the partial implementation of that model in software). For most projects, however, many changes occur just in the software as the controller design concentrates on the key building blocks providing a partial model of the system. Furthermore, the design necessarily abstracts many of the computing elements and context that must then be implemented in software. Sometimes these software changes make it back to the model through some mechanism like a bug tracker. Most often, however, implementation changes do not make it back or cannot be incorporated into the model. This causes a challenging divergence as the implementation of those abstractions in the software can have significant effects on the performance of the

control system, potentially invalidating the properties so carefully proven at the design level.

In spite of the prevalence and potential impact of this evolution, we, as a community, know very little about how the controllers that we so carefully craft change during their evolution, particularly at the software level. Our courses on control design, our textbooks, the tools we use, and the most promising research efforts *largely ignore the evolution of controllers*. We are distinctly aware of the inherent mismatches between the physical and software worlds (e.g., continuous vs. discrete, infinite vs. finite), but lack an understanding of how those mismatches manifest as the software changes.

1.2 Contributions

In this thesis, we propose processes and tools that would help control engineers to understand the controller evolution and divergence from a software perspective. It is particularly relevant to safety-critical systems that explicitly rely on model properties or implicitly rely on their assumptions. In this work we shed light on this evolution by performing a first-of-its-kind case study exclusively on control software to show how and in what way it evolves. To do this we examine 964 commits to 10 controller system files from ArduPilot and Paparazzi UAV - popular autopilot control software systems used on a wide range of Unmanned Air Systems (UAS) [2, 55]. First, to provide a baseline for how much a controller evolves, we report metrics capturing how much controller-related evolution happens in these controllers. Our results show that controllers system files changed up to 30 times a year and with enough lines of code changes to rewrite the original controller over 9 times over their lifetime. **Implication**. This means as control software matures it may have little code in common with the original, and as a result, unless a tight correspondence between model and software is enforced the evolved model

and control software may diverge drastically. This suggests that techniques such as control synthesis and model-based design techniques and tools must focus on accommodating this type of incremental process and evolution.

We then identify 4 categories that capture the evolutionary changes resulting from inherent mismatches between system models and controllers, and their software implemented counterparts. Our results show that although some changes stem from an inherent mismatch between the continuous time/space physical model and its corresponding discrete software implementation, the majority of the changes were associated with handling exceptional conditions, and with the limitations and distinct data representation of the underlying computing architecture. **Implication**. This points to an unexplored opportunity for automated synthesis and software development techniques that can bridge these mismatches appearing during software evolution that may render carefully verified model properties invalid at the control software level.

Last, we explore the effects of software evolution in the performance of 3 controllers designed with Simulink. To do that, we leverage static code analysis techniques from the software engineering discipline. Using this analysis, we created a process that would allow creation of generic templates to mimic the controller code evolution. Our analysis on 3 Simulink controllers resulted in 21 unique templates to represent the mismatches between system models and controllers, and their software implemented counterparts. This opens up opportunities for control engineers to create templates that are specific to their own controllers code evolution environment. In other words, control engineers can provide their carefully crafted mathematical model restrictions and expected fault-behavior in these templates.

In addition to this, we prototyped a tool that understands these templates and generates versions of the original code with mutated regions reflecting the categorized changes we observed in ArduPilot and Paparazzi UAV. The tool is written at a compiler

level that is capable of analyzing C family programming languages (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript). It takes as input the controller code (this is automatically generated by the Simulink toolset from carefully crafted control models) and can be configured to generate different type and different number of mutated programs. Executing the mutated programs (it is commonly referred as mutants) and observing the output results demonstrate how small and typical software changes can dramatically impact control performance.

We report on three major findings. First, control software evolves quickly and controllers are rewritten in their entirety, many times over through the controller's lifetime, which implies that the design, synthesis, and implementation of controllers must support not just the initial baseline system but also their incremental evolution. Second, many software changes stem from an inherent mismatch between the continuous time/space physical model and its corresponding discrete software implementation, but also from the mishandling of exceptional conditions, and limitations and distinct data representation of the underlying computing architecture. Third, using our mutation tool that we developed, we show that small code changes can have a dramatic effect in a controller's behavior, which implies that further support is needed to bridge these mismatches as carefully verified model properties may not necessarily translate to its software implementation. In summary, our contributions are:

- A novel approach capturing the controller evolution in software, an aspect overlooked by control community.
- Developed four categories to capture controller evolution in software: Precision and Accuracy, Exception Handling, Time and Space Model, and Resource Attributes.
- Developed a controller-specific mutation tool that mutates code generated from a model-based control design paradigm. The tool also compares outputs of the

mutated code to the original to assess the impact of software changes on control performance.

We also present possible future directions for the work presented in this thesis.

1.3 Innovations

In this thesis, we perform a first-of-its-kind study to understand the evolution of controller from a software perspective. The study is designed to highlight the mismatches between the control design and software. We also invent four mismatch categories, explain them with definition and examples. In addition to this, we make the first attempt to study the controller impact analysis using the proposed mismatch categories. We achieve this using our generic templates, mutation tool and step response characteristics. For this purpose, we created 21 unique template mutators for the three categories: Precision and Accuracy, Exception Handling, and Time and Space Model. The Resource Attributes category is excluded due to the tight dependence on specific hardware configurations. Lastly, we use a new validation technique to analyse the impact of our mutated code using step response characteristics.

Chapter 2

Related work

This chapter presents the prior work that is relevant to understand the mismatches between control design and software. Most work at the intersection of software and control has examined the impacts of the disparity between the continuous mathematical models representing physical systems and controllers and the fundamentally discrete nature of computing [68]. Such research focuses on the effects of computation (e.g., quantization, delay) on the controller and seeks to find ways to incorporate them into controller design [67]. This is the substance of digital control theory [23].

2.1 Software development and control design

The control community does not generally examine the role the software development process plays in impacting control design. But examining control software and its evolution could have far reaching impacts. For example, in the process of software maintenance, a year after the controller design, if a key calculation alters the precision by changing `fabs()` to `fabsf()`[1] does this impact system stability? Does a software change to limit memory stack size, a limitation of the computer architecture, cause a function call chain to fail, impacting controller performance? A study of control software evolution

[1] `fabs` operates on type `double` while `fabsf` operates on type `float`

can provide insight into how these effects could be mitigated either in the control model or in the software evolution process.

This motivation has led to some work focusing on software and control systems. Feron has examined how to integrate proofs of important control system properties, such as stability, directly into software [19]. This can alert the software developer when sensitive code is being modified, and provide a mechanism for verification processes to assess correctness. But unless the annotation process is less costly, the sources of unsoundness controlled, and the tools well integrated into the developer's environment and workflow, such strategies will struggle to gain mainstream acceptance [32, 9].

In safety-critical systems, model-based design strategies ideally create a 1:1 correspondence between the model and the software [58, 18, 68]. This strategy has been included in the most recent revisions of DO-178C "Software Considerations in Airborne Systems and Equipment Certification" [59] and its supplements [10]. This is done by building models in MATLAB, Simulink, Stateflow, or other tools, verifying these models, and then autogenerating corresponding code. In this paradigm the code autogeneration tool must be certified to produce provably correct results. While this strategy links the model and software it may only exist in domain-specific applications [35], and may not link 3^{rd}-party software libraries, drivers, or other specialized pieces of code used in development of the system, or may be incomplete.

2.2 Control software validation and verification

The software engineering community has developed techniques to cope with the validation and verification of systems that includes control software (e.g., [6, 47, 26, 37]), or their sound application to assist in self-adaptation [22]. Unfortunately, outside of highly regulated safety-critical systems, development use of these strategies is limited due to

high costs. This is particularly noticeable in the emerging UAS industry where open source autopilots (e.g., ArduPilot [2], Paparazzi UAV [55], PX4 [57]) are used extensively on various types of hardware with contrastingly very light regulations and rigor in design and test processes.

2.2.1 Software evolution

Software evolution has been an active research topic for decades, and the realization that successful systems evolve and how it evolves has led to laws of software evolution [41], and a rich suite of techniques to understand, handle, and support changes associated with all the entities involved in the software development process [49]. The focus of the study in this thesis is on analyzing the evolution of control software developed independently of model-based design[18], synthesis [63], or domain-specific annotations in code [19].

Following established practices [12], we analyzed two bodies of small, open-source, unregulated, safety-critical control software for which there are hundreds of available code changes recorded with commit level granularity. We have chosen these bodies of software for two key reasons. First, a large and increasing portion of critical software development with wide-reaching impacts is being developed in lightly controlled development and largely unregulated environments - such as the UAS industry. Second, understanding how the software evolves and reasoning about where mismatches with the model are likely to occur can pinpoint areas that future studies and techniques must target.

2.2.2 Mutation testing

Mutation testing introduces faults in a target software and checks if it can find errors. It is a type of software testing technique used in the verification process of a software. Existing test cases and data from unaltered software can help to find out errors after mutation testing introduces faults. Since the invention of mutation testing in 1971 [42], it

has been applied in different application domains and different programming languages. In [30], Jia and Harman provides a comprehensive survey on mutation testing. Mutation testing is a fault-based testing technique, Baker and Habli [4] applied this technique on safety-critical systems to show that the code coverage is not adequate as a criteria for DO-178B certification authorities sufficiency testing. DO-178B purpose is *"to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements"* [15]. In addition to this, in reference to safety-critical systems, mutation testing is applied in a civil nuclear software program [13].

Our mutation tool also builds on a large body of work in software engineering, and more specifically on mutation testing. Mutation testing aims to evaluate the strength of a test suite in terms of the percentage of code versions with seeded code changes it can detect. Those versions are called mutants; it is said to be alive or dead based on the presence of a change detected by the test suite between the output of the original program and the mutated program. There is a large number of mutation approaches available, as well as several analyses to improve the effectiveness and efficiency of the mutation process. For more information on the topic we refer the reader to [30]. In our setting, we utilize mutation as a way to mimic the evolution we observed in ArduPilot and Paparazzi UAV, and then to assess the impact of those mutations on simple controller operations.

Chapter 3

Controller evolution: an empirical study

In this chapter, we propose an analysis process to study the software code changes of a controller. The primary objective of this process is to answer research questions that help to understand the evolution of controller development. The approach is performed manually on two popular control software: ArduPilot and Paparazzi UAV.

3.1 Study

The following research questions will provide a foundation for understanding and characterization of control software evolution, and will underscore future tools that incorporate this knowledge into a framework for controller development:

RQ1: How does the software implementing a control system evolve? We seek to quantify the degree and nature of changes in control software in the absence of an explicit control model.

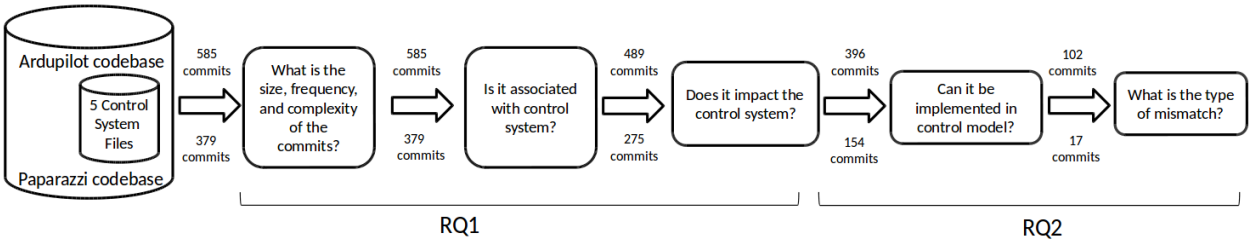


Figure 3.1: Overview of Study Analysis Process

Table 3.1: ArduPilot Files Examined

File	Type
libraries/AC_PID/AC_PID.cpp	Controller
libraries/AC_AttitudeControl/AC_PosControl.cpp	Controller & Estimator
libraries/AC_AttitudeControl/AC_AttitudeControl.cpp	Controller & Estimator
libraries/AC_WPNav/AC_WPNav.cpp	Waypoint & Navigation
libraries/AP_Baro/AP_Baro.cpp	Sensing

Table 3.2: Paparazzi Files Examined

sw/airborne/firmwares/fixedwing/stabilization/stabilization_attitude.c	Controller & Estimator
sw/airborne/firmwares/rotorcraft/stabilization/stabilization_attitude_euler_int.c	Controller & Estimator
sw/airborne/firmwares/rotorcraft/stabilization/stabilization_rate.c	Controller & Estimator
sw/airborne/firmwares/rotorcraft/guidance/guidance_h.c	Waypoint & Navigation
sw/airborne/boards/lisa_l/baro_board.c	Sensing

RQ2: To what degree can the changes in the control software be captured by a control model or constitute mismatches between the model and the software? We focus on characterizing the space of software changes that are *rarely* part of the control model.

3.1.1 Analysis Artifacts

For the purpose of our empirical study, we required artifacts that included significant control software systems with many available versions reflecting their evolution.

The first artifact is the popular ArduPilot [2][1]. ArduPilot has over five years of well maintained history that provides, among other subsystems, a sophisticated control system for autopilot support that can operate on a variety of vehicles including airplanes, multirotors, helicopters, and boats. The code is accessible through a git repository (https://github.com/ArduPilot/ardupilot) that stores the code changes committed by

[1]The ArduPilot website reports that over one million vehicles use this code base, including companies like 3DR, PrecisionHawk, AgEagle, Insitu Boeing, Kespry, branches of the US military, and NASA among others.

the developers since 2010. As of January 2018, the repository includes 347 contributors that have committed almost 30,000 changes. The latest version of ArduPilot contains approximately 200k lines of code (LOC)[2] in C/C++. We focus our analysis on the evolution of a handful of ArduPilot files written in the C/C++ programming language that are part of the core control library. We selected the core control files (see Table 3.1) that provide coverage of functionality associated with position and attitude control. We analyzed 585 commits[3], the primary unit of change we consider, where each commit included changes to at least one of the target files.

The second artifact is Paparazzi UAV [55] which has over 11 years of development history. Paparazzi UAV provides autopilot capabilities for fixed-wing, rotorcraft, and a few hybrid vehicles. The code is accessible through a git repository https://github.com/paparazzi/paparazzi. As of January 2018, the repository includes 97 contributors and ~15,000 changes. The latest version of Paparazzi UAV contains approximately 190k LOC in C/C++. We again selected control files (see Table 3.2) central to position and attitude control and analysed 379 commits.

Table 3.3: Definitions of Categories for Mismatches Between Models and Software

Category	Definition
Resource Attributes	A software change resulting from features or limitations of the computing architecture, including software and hardware. Such changes are often intended to better fit or utilize existing resources such as memory, energy, or bandwidth.

[2]LOC - Lines of Code - is a count of lines in the text of source code excluding comment lines [51].

[3]In the version control system git, a commit consists of one or more changed files identified as a single change unit by the developer and assigned a single identification number by git.

Precision and Accuracy	A software change that modifies a measured value or a numerical calculation in order to more closely mimic continuous mathematics. Such changes often consist of utilizing improved functions in advanced math libraries or newer sensor devices, and simply using types with more bits for representation.
Time and Space Model	A software change resulting from the intrinsic discrete nature of the computing system in representing time and space. Such changes often consist of handling the inherent mismatch between continuous and discrete paradigms in representing and manipulating time in the calculations of derivatives and integrals, in the manipulation of variables associated with the vehicle location or motion, or in governing the periodic execution of certain pieces of code (e.g., tasks).
Exception Handling	A software change resulting from the handling of anomalous conditions that would otherwise result in computational failures. Such changes often consist in additional support for conditions to adhere to either mathematical laws (e.g., dividing by zero), or computational laws (e.g., unexpected input, seg fault, etc.).

Neither ArduPilot nor Paparazzi UAV have formal models of the controller, and as a result do not practice complete model-based design. These controllers are maintained and modified primarily in software. This is common practice among small companies,

researchers, and hobbyists in areas not subject to strict regulation and certification requirements. Because ArduPilot and Paparazzi UAV provide safety-critical software to unmanned systems without a rigorous certification/verification process their software is an excellent example of control software development that may be (at best) weakly linked to a mathematical model with provable guarantees.

Table 3.4: Examples of Categories for ArduPilot Mismatches Between Models and Software in ArduPilot

Category	Examples

Resource Attributes	This change stores variables in flash memory
	instead of static random access memory.
	`Commit id: 452749149fd4d3e910e6ed22a6f861d5862a4b0`
	`Committers comment: convert AC_PID library to AP_Param`
	`...`
	`+const AP_Param::GroupInfo AC_PID::var_info[] PROGMEM={`
	`+    AP_GROUPINFO("P",    AC_PID, _kp),`
	`+    AP_GROUPINFO("I",    AC_PID, _ki),`
	`+    AP_GROUPINFO("D",    AC_PID, _kd),`
	`...`

Precision and Accuracy	Replaces `fast_atan` with `atanf` to improve accuracy and precision for calculating the target pitch angle.
	`Commit id:`
	`↪ 872583f4412ade16a31e8b7bd0363c294a20d301`
	`Committers`
	`↪ comment:AC_AttitudeControl`
	`↪ removed fast_atan`
	`...`
	`-_pitch_target = constrain_float(fast_atan(`
	`↪ -accel_forward/(GRAVITY_MSS * 100))*`
	`↪ (18000/M_PI_F),-lean_angle_max,lean_angle_max);`
	`+_pitch_target = constrain_float(atanf(`
	`↪ -accel_ forward/(GRAVITY_MSS * 100))*`
	`↪ (18000/M_PI_F),-lean_angle_max,lean_angle_max);`
	`...`

Time and Space Model	This change alters the time representation from seconds to milliseconds to more frequently check the position controller activity.
	`Commit id: 88ec13b10d913d72cdb0b24ba2e1244e6ed37734`
	`Committers comment: fix build`
	`...`
	`-  if (dt > POSCONTROL_ACTIVE_TIMEOUT_SEC) {`
	`+  if (dt > POSCONTROL_ACTIVE_TIMEOUT_MS*1.0e-3f) {`
	`...`

Exception Handling	This change checks whether the input variable to the PID controller is infinite or undefined before using it to calculate the PID terms of the controller.
	`Commit id: ae77c18a1933dcb00eb9fc838872119b2250915c`
	`Committers comment: Input to the PID controller is protect against NaN and INF.`
	`...`
	`+ // don't pass in inf or NaN`
	`+ if (isfinite(input)){`
	`...`

3.1.2 Analysis Process

The process is summarized in Figure 3.1 and consists of a set of filtering and analysis steps for each of the questions. The process to answer **RQ1** starts by systematically querying the git repositories to quantify the degree of change on the target files in terms of size, frequency, and people involved. To do this we downloaded the latest repositories and developed a set of scripts, in combination with the git client management tool Giteye, to collect the data.

To better understand the nature of the changes we also devised a procedure to identify commits that are most likely associated with changes that will impact the control system. This procedure focused on the developers' comments and code changes, and was partially automated through a syntactic file search using common control keywords (e.g., control, derivative, error, feedforward, filter, frame, frequency, gain, integral, kalman, proportional) and also keywords specific to the target autopilot controller (e.g., acceleration, altitude, distance, pitch, roll, yaw, waypoint, speed, velocity). This process also took into consideration the online documentation explaining the roles of key

configurable parameters and variable naming practices.

Table 3.5: Examples of Categories for Paparazzi UAV Mismatches Between Models and Software in Paparazzi UAV

Category	Examples

Resource Attributes	The horizontal feedforward gain is defined as 0. This is later used for multiplication bit operation to determine the control command for horizontal guidance navigation. Bit representations of control variables cannot be represented in the control model.
	`Commit id: 5de51d35588fa0080db7b8416924a900b405b4e9`
	`Committers comment: [guidance] fix IGAIN precision and add VGAIN based on #682 this may introduce too large horizontal guidance IGAIN in rotorcraft airframe files`
	`...`
	`+#ifndef GUIDANCE_H_VGAIN`
	`+#define GUIDANCE_H_VGAIN 0`
	`+#endif`
	`...`
	`guidance_h_cmd_earth.x = pd_x +`
	`((guidance_h_vgain * guidance_h_speed_ref.x) >> 17) +`
	`((guidance_h_again * guidance_h_accel_ref.x) >> 8);`
	`...`

Precision and Accuracy	Replaces `int32` with `float` to improve accuracy and precision for calculating the angular rate set point.
	`Commit id: 0c95b9e26edaba085f210b41d0a8325b607d9ada`
	`Committers comment: [rotorcraft] converted PI rate controller to floating point  closes #1624`
	`...`
	`- struct Int32Rates stabilization_rate_sp;`
	`+ struct FloatRates stabilization_rate_sp;`
	`...`

Time and Space Model	This change alters the execution frequency of the navigation task from 10 Hz to 16 Hz.
	`Commit id: 624ce9eea923bff55e3c913363e9b42fe9cd6aab`
	`Committers comment: navigation function in guidance; frequency set at 16 Hz`
	`...`
	`-  RunOnceEvery(50, nav_periodic_task_10Hz());`
	`+  RunOnceEvery(32, nav_periodic_task());`
	`...`

Exception Handling	This change prevents a divide by zero error by ensuring the variable is greater than zero before being used to calculate the navigation ratio for the vehicle controller.
	`Commit id:`
	`↪ 7f91efa2854fee702a6601256dea5ff195e58f80`
	`Committers comment: Fixed Error`
	`↪ preventing AGR climb`
	`from working. Navigation would`
	`↪ not blend.`
	`...`
	`+if ( AGR_BLEND_START > AGR_BLEND_END &&`
	`↪ AGR_BLEND_END > 0){`
	`...`
	`+nav_ratio = AGR_CLIMB_NAV_RATIO + ( 1 -`
	`↪ AGR_CLIMB_NAV_RATIO)*(1 -`
	`↪ (fabs (altitude_error) - GR_BLEND_END) /`
	`↪ (AGR_BLEND_START - AGR_BLEND_END));`
	`...`

The resulting commits (489 for ArduPilot and 275 for Paparazzi UAV) were further analyzed to discriminate between changes deemed semantically equivalent such as those caused by documentation, refactoring, or abstraction meant to ease the maintenance of the software without directly impacting the functionality. For example, code found to be repeated may be extracted into a function call. This, theoretically, has no impact on the controller as it is purely a software maintenance change. This filtering left 396 ArduPilot and 154 Paparazzi UAV commits/changes impacting the controller directly.

The process to answer **RQ2** (see Figure 3.1) filtered the remaining commits by making a qualitative analysis to determine whether the change could have been handled in a typical control model. Again we note that neither ArduPilot nor Paparazzi UAV have

formal control models, so our assessment consists of a conservative judgement of whether, if a mathematical model of the control system would be available, such a model could accommodate a given change. It is conservative in that, when in doubt, we assume that a control model could handle such a change. More specifically, unless the changes that are: 1) tightly associated with the computing architecture, 2) the representations of data in that architecture, 3) the discretization of time and space to function in that architecture, or 4) the handling of anomalies due to that software functions, we assume it could be represented in a control model. When we determine that a control model would not typically include such a change because it is tightly associated with the computing software context, we assume it constitutes a mismatch between model and code that could have an impact on the system behavior. We then proceed to classify each change into one of four categories that emerged as we analyzed these mismatches and grouped them according to their characteristics, defined in Table 3.3 and examples are provided in Table 3.4 & 3.5. This classification procedure was costly, with some changes requiring minutes and others requiring hours and the participation of all authors. Furthermore, this classification process was iterative as new mismatches emerged that either did not fit existing categories or fit multiple ones.

3.1.3 Threats to validity

This study has shortcomings that may impact the validity of the findings. First, the scope of the study is limited to the software side of controller evolution. This choice was intentional and allowed us to quickly leverage readily available data while decoupling the evolution occuring in software from that which would occur in a model. This is the first step in this line of work, and studies of controller design evolution, controller design coupled with control software, and impact on system performance will provide a broader understanding of the topic.

Second, our study is focused on a subset of files of two control software systems. This choice was opportunistic in that Ardupilot and Paparazzi have been widely deployed, so findings in these code bases can still be valid for similar systems (e.g., LibrePilot [56], PX4 [57]). Likewise, even though the cost of analyzing hundreds of commits limited us to study ten files, those files perform different controller tasks and were designed by different groups of developers. As a result, we anticipate these findings will also apply to other files designed by other developers. We also ackowledge that the granularity of change we studied, commits, may not expose all code changes made by developers.

Third, our analysis had a quantitative aspect that is partially automated and higly reproducible, and a qualitative aspect that in many instances required us to make judgement calls. Such judgement calls are subject to many biases, which we tried to reduce by defining clear criteria for filtering and classification, by having multiple authors check different parts of the results, and by iterating and revisiting the results as anomalies emerged. We have prepared a package with the detailed data for others to review our choices[4].

3.2 Results

We now present the results for our study, answering **RQ1** and **RQ2** described in Section 3.1.

3.2.1 Answers to RQ1 - How Much do Controllers Evolve?

We quantify the evolution of the selected ArduPilot and Paparazzi UAV control files in Table 3.1. Results are captured in Table 3.6 & 3.7 showing the evolution of the software. It reports on initial and final LOC, # of commits, LOC changed, and people involved for

[4] https://nimbus.unl.edu/CE/controllerevolution.html

Table 3.6: Overview of ArduPilot Control Software Evolution

Filename(→)	AC_PID.cpp	AC_PosControl.cpp	AC_AttitudeControl.cpp	AC_WPNav.cpp	AP_Baro.cpp	Total
Date of earliest commit	1/28/2012	2/14/2014	2/14/2014	4/13/2013	6/27/2012	
Date of the latest commit	2/18/2017	4/27/2017	6/22/2017	7/9/2017	7/7/2017	
LOC in the earliest commit	54	601	152	166	55	1028
LOC in the latest commit	141	661	440	754	382	2378
Commits involving that file	37	134	127	185	102	585
LOC changed in those commits - code churn	463	1672	3350	3252	1043	9780
People involved	6	3	3	4	6	8
Growth (%)	161.11	9.98	189.47	354.22	594.55	131.32
Rewrite Rate	8.57	2.78	22.04	19.59	18.96	9.51

each of the files of interest. Changes to these key files were made by 28 developers who changed 15,066 LOC over 964 commits throughout the lifetime of the files. The guiding principle in this analysis is to examine the evolution of control software, and as a result, throughout the presented results we focus on *changes* to the software which excludes the first commit representing the initial implementation.

The metric in row 8 of Table 3.6 & 3.7 assesses how much the software grows over its lifetime. Growth is computed as $\frac{(X-Y)}{Y}\%$ where X is the number of lines of code, excluding comments, in the latest commit (row 4 in Table 3.6 & 3.7) and Y is the number of lines of code (excluding comments) in the earliest commit (row 3 in Table 3.6 & 3.7). Growth captures the net lines of code changed including changes stemming from model

Table 3.7: Overview of Paparazzi UAV Control Software Evolution

Filename(→)	stabiliz-ation_at-titude.c	stabiliz-ation_at-titude_e-uler_int-.c	stabiliz-ation_ra-te.c	guidance-_h.c	baro_boa-rd.c	Total
Date of earliest commit	10/19/06	07/26/09	02/10/09	02/10/09	08/21/10	
Date of the latest commit	02/19/17	03/22/16	04/27/16	12/23/17	12/27/17	
LOC in the earliest commit	135	89	36	126	77	463
LOC in the latest commit	323	195	150	546	168	1382
Commits involving that file	72	52	60	159	36	379
LOC changed in those commits - code churn	707	636	757	2859	327	5286
People involved	10	5	5	12	5	20
Growth (%)	139.26	119.1	316.67	333.33	118.18	198.49
Rewrite Rate	5.24	7.15	21.03	22.69	4.25	11.42

clarifications, new features, bug fixes, and software maintenance. As an example for this metric, `AC_PID.cpp` had 54 lines of code initially, and in the latest commit has 141 lines of code, a growth of 161%. The ArduPilot files have an average growth rate of 131% while the Paparazzi UAV files average growth rate is 198%, implying that the initial implementations required significant changes to complete them and refine them, and more generally that these control files, like any successful software, grow in complexity as they evolve. In some cases, like for `AP_Baro.cpp`, we notice a dramatic growth of almost ~600% to abstract common features, support more devices, and improve calibration. Other files like `AC_PosControl.cpp` exhibit a more stable development from the start with only ~10% growth.

Growth does not, however, capture the amount of change occurring in a file. To measure this, code churn is defined as the total number of lines of code changed (row 6 in Table 3.6 & 3.7) [24]. For example, the code churn for `AC_AttitudeControl.cpp` is 3350 lines of code with an average of over 26 lines changed per each of its 127 commits. To further emphasize the seriousness of code churn for control software we use a metric we call "Rewrite Rate" that captures how many times the original controller has been essentially rewritten from a software perspective. We use $\frac{Z}{Y}$, where Z is the total number of LOC changed in row 6 in Table 3.6 & 3.7. However, high growth does not necessarily mean high churn. `AP_Baro.cpp`, for example, exhibits the highest growth of all files, but `AC_AttitudeControl.cpp` shows the highest code churn. Of the ten files, five have Rewrite Rates ~20 indicating those **control files have almost nothing in common with the original versions**. To give perspective, even the file with the lowest rate, `AC_PosControl.cpp`, has been rewritten almost three times.

For software engineers this evolution is not necessarily surprising as it mimics what is seen in other evolving system files. For control designers, however, this implies that a **controller implemented in software may significantly diverge from the original design without a correspondence to the model unless those ties are continuously enforced**. It also means that if a tight correspondence between the model and software is not enforced, a large amount of time must be spent updating the controller to correspond with the software (dashed arrow in Figure 1.1) or most likely the model will become obsolete along with its proven guarantees.

Finally, we observed a high concentration of changes in a smaller group of files. Two thirds of the code churn in ArduPilot occur in two files, and a similar change concentration is found in a single Paparazzi UAV file. **We conjecture that files like `AC_PID.cpp` containing some key abstractions may "settle" into a steady state as other software modules come to depend on core functionality.** Such functionality with higher

Table 3.8: Classification Results of ArduPilot Mismatches Between Models and Software

Category(↓) / Filename (→)	AC_PID.cpp	AC_PosControl.cpp	AC_AttitudeControl.cpp	AC_WPNav.cpp	AP_Baro.cpp	Total
Resource Attributes	1	2	0	1	0	4
Precision and Accuracy	7	15	7	13	9	49
Time and Space Model	2	15	3	11	8	37
Exception Handling	4	6	2	12	5	29
Total Commits With Mismatches	12	29	11	32	18	102

Table 3.9: Classification Results of Paparazzi UAV Mismatches Between Models and Software

Category(↓) / Filename (→)	stabilization_attitude.c	stabilization_attitude_euler_int.c	stabilization_rate.c	guidance_h.c	baro_board.c	Total
Resource Attributes	0	0	0	3	0	3
Precision and Accuracy	0	1	2	5	0	8
Time and Space Model	0	0	0	1	1	2
Exception Handling	1	1	0	5	0	7
Total Commits With Mismatches	1	2	2	11	1	17

stability may constitute more cost-effective targets for modeling and verifying more extensively at design time, before transferring them into software.

3.2.2 Answers to RQ2 - What Evolution Results from Model and Software Mismatches?

If control models and software evolve independently then it is critical to understand what kind of changes prevent a 1:1 correspondence between them. We classified the 102 ArduPilot commits and the 17 Paparazzi UAV commits from the last stage of Figure 3.1 into the four categories defined in Table 3.3. These categories represent the primary mismatches resulting from the incongruences between control models of the physical system and the computational paradigm of software implementation. In the right hand

column are examples to clarify the types of changes in these categories.

The mismatched commits and classifications are tallied in Table 3.8 & 3.9. Each commit could have an arbitrary number of LOC changed, and hence a single commit may have multiple mismatches and be classified into more than one category.

Overall, the distribution of mismatches is similar across ArduPilot and Paparazzi UAV (see "Total" columns in Table 3.8 & 3.9) However, the number of mismatches in ArduPilot is five times larger than Paparazzi UAV despite having smaller growth, rewrite rate, and fewer developers involved (from Table 3.6 & 3.7). This is due, in part, to the larger number of commits that affect the control model in ArduPilot. Further explanations may be that, in Paparazzi UAV, some control elements were externalized into a separate configuration file to isolate potential changes to the system. The analysis of such files are left for future work.

We also observe that the number of mismatches per file is correlated with code churn exhibited by the file, with `AC_AttitudeControl.cpp` and `guidance_h.c` being the most affected. Still, `AC_AttitudeControl.cpp` seems to be the exception, suggesting that other software engineering factors (e.g., abstractions, refactoring) likely contributed to the evolution changes for `AC_AttitudeControl.cpp`. Generally, however, mismatch changes tracks proportionally with total number of changes.

A comparison between `AC_AttitudeControl.cpp` and `AC_WPNav.cpp` in ArduPilot reveals that despite having roughly similar starting code size and total LOC changed in their lifetime, `AC_AttitudeControl.cpp` has only 30% as many mismatch changes. `AC_PosControl.cpp` and `AC_WPNav.cpp` have similar mismatch changes even though `AC_WPNav.cpp` was initially much smaller but grew to be twice as large and have much higher code churn. This is not surprising as `AC_WPNav.cpp` is the navigation code library that calculates the desired velocity, and acceleration to reach the destination. When the user provides the destination origin, `AC_WPNav.cpp` creates a flight path using spline waypoints and

ensures the vehicle operates within the set range of acceleration, velocity, and speed. It also determines whether the vehicle has reached its target to within a certain radius. Such a software module is critical and difficult to develop correctly due in part to the many calculations requiring many vehicle and environmental parameters. Supporting this conclusion is a similar observation for `guidance_h.c` in Paparazzi UAV given its high relative mismatches, churn, and growth compared with other Paparazzi UAV files. This is perhaps the apex of joint model and software integration.

Observing the categories, "Precision & Accuracy" was the biggest source of mismatches between model and software (see "Total" column in Table 3.8 & 3.9), accounting for 48% of the ArduPilot and 47% of the Paparazzi UAV mismatches. This implies developers prioritized improvements to the precision and accuracy of calculations to either 1) more closely mimic continuous mathematical assumptions of infinite precision, or, 2) prioritize improvement in computational system performance while sacrifing precision and accuracy. We observe that some of these changes were not particulary complex (changing an `int` to `float`), while others involved utilizing special functions from a math software library. Still others, like switching `fabs` to `fabsf`, seem to sacrifice precision presumably to be consistent in the use of `float` to represent decimals and avoid unnecessary conversions potentially saving unnecessary computations at runtime. These mismatches were pervasive throughout the evolution of all files.

"Time and Space Model" mismatches are concerned with accounting for and tracking discrete time in control software. While we considered discretized space in the same category, which would be more prominent in control software incorporating, for example, a computer vision component, we did not observe any discretized space mismatches in this set of files. Ensuring consistency between periodic execution of a controller and associated computation of discrete derivative and integral equivalents is critical for correct control performance. We observed many changes that focused on improving this

consistency in a programming language (C/C++) that does not natively provide semantic support for timing [39]. Most of these mismatches occur in the navigation/guidance (`AC_WPNav.cpp` and `guidance_h.c`), and position controller (`AC_PosControl.cpp`) portions of the controller software. Our results reporting on the number of changes involving timing provide further support for Lee's claims that timing in computation is a major obstacle to the development of combined cyber-physical models in which determinism is preserved [39]. Although many of these mismatches could be incorporated into the model by using MATLAB toolboxes such as "TrueTime" [25] or checked using other timing verification strategies like UPPAAL [38], these are often costly and continue to be underutilized in many development environments like the one we have studied.

Often overlooked by control designers are the undefined mathematical operations in engineered systems such as dividing by zero, or multiplying by ∞. In mathematical models these exceptions are built into the assumptions of continuous mathematics and are *implicitly* avoided. In software they must be *explicitly* avoided with lines of code protecting potentially undefined operations from causing the program to end prematurely or perform incorrectly. This exception handling also extends to software and computing architectural rules that must be obeyed (e.g., handling `NULL` pointers). The combined 36 total mismatches in this category (row 4 of Table 3.8 & 3.9) suggest that even software developers may take implicit assumptions about exception handling for granted. As the code evolves these exceptions are dealt with possibly in response to failed test cases or bug reports.

Finally, computing architectural issues result in some mismatches we classified as "Resource Attribute." Modern programming language abstractions have helped reduce these mismatches as compilers and libraries allow flexibility and optimizations without special programmer knowledge, and operating systems provide virtual memory and thread handling for executing processes. The small number of mismatches in this category

(row 1 of Table 3.8 & 3.9) is likely a result of the non-specialized hardware platform that ArduPilot and Paparazzi UAV run on. Had the control software required a specialized Digital Signal Processing (DSP) chip, or Graphical Processing Unit (GPU) we would have expected to see more mismatches in this category to accommodate those special-purpose computing architectures. Nevertheless, this category represents an important side-effect of software implementations of controllers - unless the control model explicitly captures the details of each target hardware architecture, programming language, 3^{rd}-party library or hardware driver, and operating system there will likely be mismatches between the model and implementation.

3.3 Conclusions

In this chapter, we studied and presented the controller evolution, through 964 commits with 15,066 control software lines changed, of two dominant open-source control software suites, ArduPilot and Paparazzi UAV. We found that control software evolves quickly; we observed an average growth of 131% in ArduPilot files and 198% in Paparazzi UAV files. We also found that amount of change occurring in a file through code churn metric, half of the files have rewrite rates of ~20 indicating control files have almost nothing in common with original versions.

In addition to this, we categorized the controller evolution into four categories. These categories represent the incongruences between control models of the physical system and the computational paradigm of software implementation. The evolution in terms of these categories revealed that "Precision and Accuracy" category was the biggest source of mismatches. "Time and Space Model" had the next highest source of mismatches "Exception Handling" category followed it. These are categories that are often underuti-

lized or overlooked. ”Resource Attribute” category represented very few but significant changes that deals with computing architectural mismatch issues.

Chapter 4

Impact of Software Changes on Control Performance

In this chapter, we introduce a tool developed to help mimic and analyze the software controller evolution and its impacts. For this purpose, we designed a tool based on mutation testing. We first define the process of mutation testing, then explain the architecture of our proposed mutation tool followed by the methodology, and discuss the divergence result impacting the controller performance.

4.1 Mutation testing process

Mutation of software is a practice used in the software engineering community to test the robustness of software and tests to small, isolated changes in the software [30]. Figure 4.1 illustrates the genericprocess ofmutation analysis, in the context ofcontroller software. In mutation analysis, from an original source program C (in our case, it is the software program of a controller), an altered program M called mutant is generated by altering the code. For example, consider a software system that has a line of code performing an arithmetic operation, a change could involve altering an addition operation of two variables into a subtraction operation of two variables. Here, the original program is performing the addition operation whereas the mutated program is performing the subtraction operation.

In the next step, we design a test oracle, the dashed rectangle represents this in Figure 4.1. A test oracle is a mechanism that would execute a test on the program to determine if they pass or fail the test. Following our example of an arithmetic operation, an oracle would provide two numerical numbers to both the programs, original and mutant. To check if they pass or fail the test, the oracle has to compare the output values from the original program and the mutated program. If the values are same, then it passes the test otherwise failed.

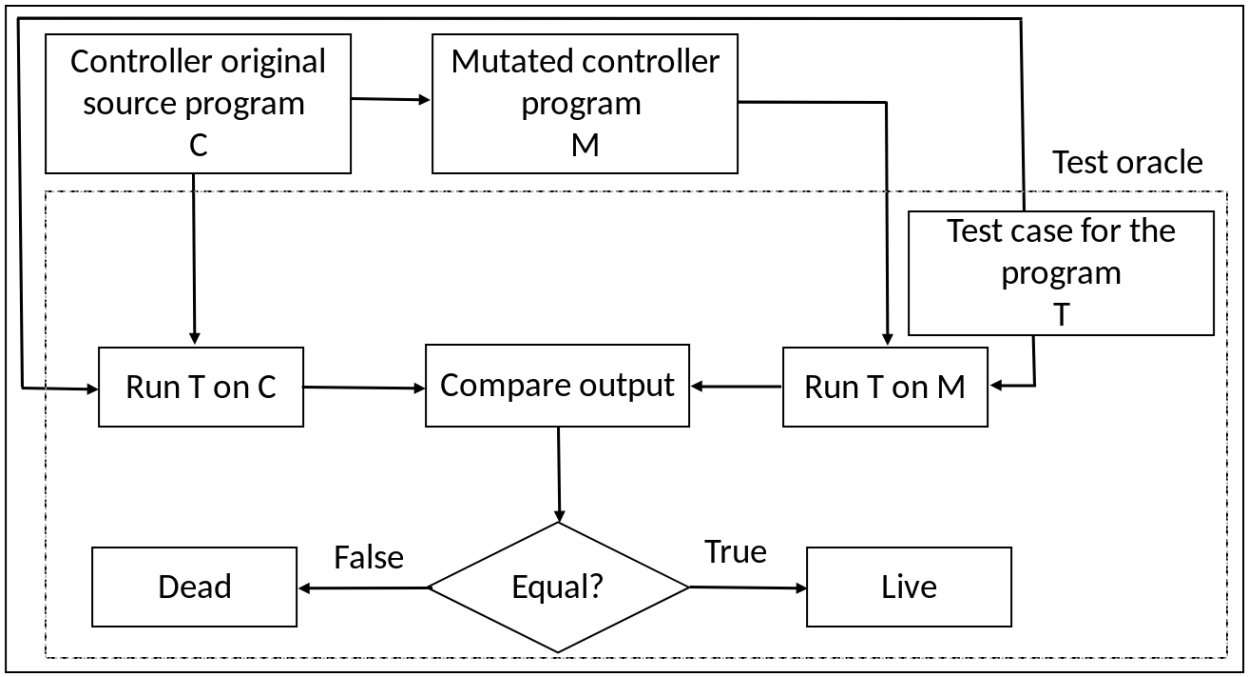


Figure 4.1: Mutation testing process overview.

Lastly, we call a mutant as live if it passes the oracle's output comparison test otherwise we call the mutant as dead. This process is repeated for all the mutants generated. The test oracle usually comprises a set of test cases such that it will test all the components of a software, in our case, the test oracle tests the behavior of the program using step response characteristics.

In traditional mutation testing [30], a live mutant is one in which the test oracle successfully did not catch a difference in the output. A dead mutant or a mutant is said to

be killed if the test oracle was able to detect a difference in the output. In this scenario, a dead mutant implies the test case is sufficient to catch errors or vulnerabilities. A perfect test suite will kill all mutants. A live mutant indicates the test was insufficient to catch the errors. We adopt this convention with the exception that we assume our test suite is ideal, and hence 'live' mutants (i.e., output of mutated code is equivalent to original code) are desirable as they imply that the controller is robust to software changes. 'Dead' mutants, in our scenario, suggest the controller was vulnerable to small software changes since the output of the mutated control code was different than the original.

4.2 Mutation tool overview

The proposed mutation tool, an overview of which is in Figure 4.2, generates code from Simulink models, mutates the code, compiles it, executes a test suite, and compares the output to the output of the original design. In the Figure 4.2, the dotted line represents our contribution, it comprises software functionality developed for this thesis. The first step is to compile and execute a Simulink model to get the output, this we will refer as original output. Next, we generate C code from the same model and make very few changes in this C code. This is called the mutated code. We repeat this process with different changes each in different locations of the C code to get different versions of mutated code. Lastly, each mutated code is executed to get output that is compared with the original output.

Our test suite for each mutant is a step response characterized by key control design quantities. The tool mutates the generated code according to our categories shown in Table 3.3. We are primarily concerned with studying the mismatches that occur between control models and control software. As a result, the tool is focused on mutating code generated from Simulink because: i) model-based design is an increasingly important,

but relatively little studied methodology, and ii) model-based design should maintain a 1:1 correspondence between the design and generated code, but many changes that *could* be made in the code may not be represented in the design.

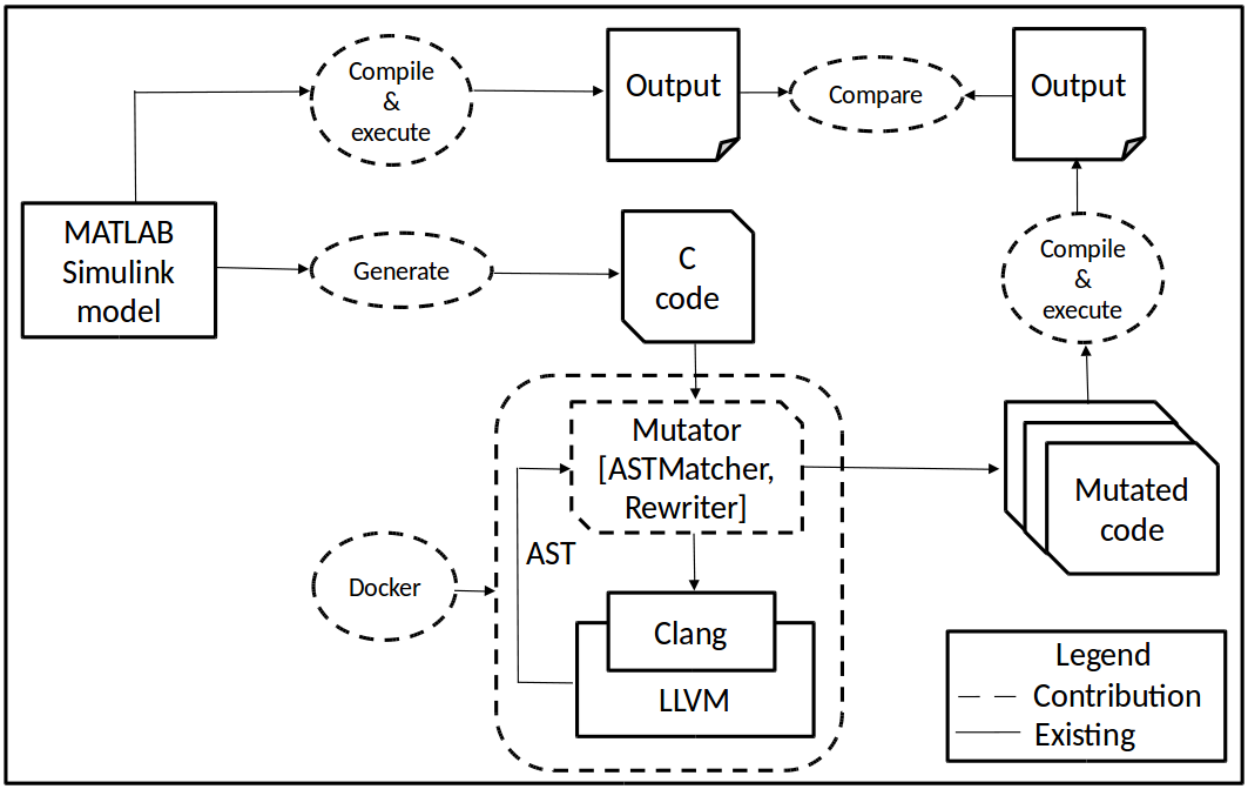


Figure 4.2: Mutation Tool Architecture.

The mutation tool can understand C family programming languages. It uses the abstract syntax tree (AST) [50] of the respective programming language to construct a tree model of the source code. AST captures the abstract syntactic structure (i.e., abstract syntax consists of a structure of data) of source code, in a tree representation, written in a software programming language. For example, consider an if-condition-then statement from a source code, the syntactic construct will be represented as a single node with three branches in the tree model. We use Clang 3.8.2 and LLVM 4.0 for this purpose and use the `MatchFinder` class of Clang to process the AST. To support repeatability, we have built this software infrastructure inside a operating-system-level virtualization, called docker

(version 1.13.1).

To study the effects of the mutated software code, we used three different, increasingly complex system design models developed in Simulink - an automotive cruise control, a helicopter, and a Boeing 747. The automotive cruise controller contains 14 MATLAB blocks, is publicly provided and made available by the University of Michigan, Carnegie Mellon University and University of Detroit Mercy [52]. The helicopter system contains 40 blocks, and is provided by MATLAB [45]. It simulates hovering conditions of a helicopter model. The third system is an airspeed and altitude controller for a Boeing 747 containing 465 blocks. This model is maintained by Michael S. Selig, Rob Deters, and Glen Dimock at the University of Illinois Urbana-Champaign [8].

4.3 Methodolgy

In this section, we discuss the methodology to create the software mutation tool using the knowledge from our previous study and the knowledge of control theory. The mutation tool has three phases and the implementation details are depicted in Figure 4.2.

4.3.1 Phase I

In the first phase, C code is generated from the Simulink model and an AST is generated from the code. The AST is parsed to identify the locations where the code could be mutated. Mutation templates, software abstractions of our mismatch categories in Table 3.3, are used to identify where code can be mutated. The templates are constructed such that many locations in the code can be mutated by a single template in a variety of ways. The tool randomly chooses a location and applies the mutation.

4.3.1.1 Template creation

Template creation is a manual process to create generic abstractions from code changes. The goal is to create generic templates that could inject code changes in any software control system. A template could inject more than one similar code variations in more than one code location. For example template d*→ d.of, will make code changes of 2→ 2.of and also 100→ 100.of that is present in different code locations. For each of these code change we can then observe the impact on the controller performance. In our case, we create generic templates, for the purpose of application we the MATLAB environment.

For creating the templates, we first identified the top most changes in each category. Some software code changes occurred very frequently and was repeated across files, while some other code changes were less frequent. So in our template creation process, we first tried to capture this high frequency changes. After gathering these changes, we wanted to create templates that are generic. For abstracting out these changes we grouped similar code changes and started creating template for them. We list these top changes and abstractions that are grouped by similarity for *Precision and Accuracy, Exception Handling*, and *Time and Space* in table 4.1, 4.2, and 4.3 respectively. This process worked well for three categories: Precision and Accuracy, Exception Handling, and Time and Space Model.

This process did not construct templates for *Resource Attributes* category because the software code changes were specific to hardware dependencies. For example, the software evolutionary code changes with a tightly coupled microprocessor configuration capabilities like processing power (8-bit or 16-bit) and memory access was a difficult challenge while constructing templates. To reproduce this in MATLAB, MATLAB provides an option to run the Simulink with different hardware specific platforms. We identified the resource attribute blocks manually. We used the MATLAB option to generate C code

Before code change	After code change	Number of occurrences
	#include<AP_Math.h >	4
hal.scheduler ->millis ()	AP_HAL:: millis ()	3
AP_Math::is_zero(_filt_hz)	is_zero (_ filt_hz)	3
!is_zero(_ki)	! AP_Math :: is_zero (_ ki)	3
0	0.0f	6
200	0.2f	2
2	2.0f	2
fabs	fabsf	4
abs	fabs	1
Int32_t	float	3
int16_t	float	2
atan2	atan2f	1
atan2f	fast_atan2	1
fast_atan2	atan2f	1
fast_atan	atanf	1

Table 4.1: 5 top most changes from Precision and Accuracy category grouped by similarity

Before code change	After code change	Number of occurrences
	Added !is_zero condition in IF	3
	accel_z_cms <= 0.0f	3
	_wp_accel_cms <= 0	2
	isnan	2
	isinf	2
isfinite	!isfinite	1
is_zero	isnan	1
is_zero	isinf	1
remove <=0.0f		2
if (_track_length == 0.0f)		1

Table 4.2: 5 top most changes from Exception Handling category grouped by similarity

for 32 bit and 64 bits systems. The resulted code was very similar to each other and hence we could not create *Resource Attributes* category templates. We are working towards getting these things done as future work. In total, we have implemented 21 unique

Before code change	After code change	Number of occurrences
	uint32_t now = hal.scheduler->millis();	11
	void AC_PosControl::set_dt_xy (float dt_xy)	5
dt >= 1.0	dt >= 1.0f	1
dt >= 1.0	dt >= 0.2f	1
if (fabsf(_alt_offset - _alt_offset_active) >0.1f) {	if (fabsf(_alt_offset - _alt_offset_active) >0.01f) {	1
POSCONTROL_ACTIV-E_TIMEOUT_MS	POSCONTROL_ACTIV-E_TIMEOUT_MS*1.0e-3f	1
float dt = (now - _last_update_xy_ms) / 1000.0f;	float dt = (now - _last_update_xy_ms) * 0.001f;	1

Table 4.3: 5 top most changes from Time and Space Model category grouped by similarity

template mutators for the three categories: Precision and Accuracy, Exception Handling, and Time and Space Model as given in Table 4.4, 4.5, and 4.6 respectively. The Resource Attributes category in Table 3.3 is excluded due to the tight dependence on specific hardware configurations.

4.3.2 Phase II

In the second phase the tool compiles and executes the mutated code to obtain a step input response from the mutated code to compare against the original model. We quantify the control performance via 8 traditional control step response quantities: rise time, settling time, settling min, settling max, overshoot, undershoot, peak, and peak time [54]. For compilation process, the tool requires us to make two configuration settings. First, the tool needs to understand the software environment and software dependencies. It requires this because the compiler requires this information to generate abstract syntax tree for the

targeted software code. Second, modify the control input values in your control software. The tool compiles and executes the target control software assuming that the control input values are changed, as needed my the user. Currently, the tool lacks a placeholder to mention the location of control input value present in the software code, this will be incorporated in the future.

4.3.2.1 Matlab specific mutation

For the purpose of generating mutants, in the third category - *Time and Space Model*, we had to improvise our process. Reason being, they involved software code changes with time and space related variables specific to ArduPilot and Paparazzi UAV. To identify the variable types and names related with time and space in MATLAB environment, we gathered all the time and space related blocks from MATLAB library manually. It is important to note here that this can be replicated for any other libraries or any other platforms, in our case this could be used in C and C++ programming languages. Also our tool has this information in separate configuration file that the user can easily modify this properties without even changing the code.

We constructed a basic Simulink model environment for these blocks and then generated C code for each of the models separately. Some models did not successfully get executed due to the requirement and dependency on other blocks, while most others got successfully executed and generate C code from them. We then use this information to identify time related variables for MATLAB generated C code, automatically. We automated this process with the help of cppchecker [43]. The cppchecker is a static analysis tool that helps to identify all the variables from the C code. We list these variables for all the identified Simulink model and hand pick those that are present in more than one models. This way we ensured that they do not bias the variable names. We then used these data types and names to come up with templates for mutating in MATLAB

environment.

4.3.3 Phase III

Finally, in the third phase we verify our results. For this purpose, we check whether the mutated code is stable, unstable, or has altered performance. The tool uses a comparison mechanism to check the step response characteristics value of the mutants with the original system. It is important to note here that each system has a different control input parameter: cruise control system has speed, helicopter system have pitch angle and roll angle, and Boeing 747 system have airspeed and altitude. In turn, each of these control input has different step values consisting of numerical quantity and a physical unit. Our verification process took these things into account and computed the absolute difference of 8 traditional control step response quantities between the mutated code and original system. Based on this difference, we classify the mutated code as stable, unstable, or has altered performance.

Table 4.4: Mutation output details for Precision and Accuracy category in all three systems

Mutation Operator	Cruise control			Helicopter			Boeing 747		
	# mutants	# compiled	# executed	# mutants	# compiled	# executed	# mutants	# compiled	# executed
int_T → uint32_T	2	2	2	6	6	6	3	3	3
int_T → real_T	2	1	1	6	1	1	3	0	0
uint32_T → int_T	0	0	0	0	0	0	3	3	3
real_T → int_T	1	1	1	3	3	3	46	46	46
d* → d*.of	75	74	74	109	108	108	263	258	258
d* → d*.o	75	74	74	109	108	108	263	258	258
d*.of or d*.o → d*	85	85	84	125	125	124	144	144	142
float F() → (double) F()	0	0	0	0	0	0	0	0	0
double F() → (float) F()	0	0	0	0	0	0	37	37	37

Table 4.5: Mutation output details for Exception handling category in all three systems

Mutation Operator	Cruise control			Helicopter			Boeing 747		
	# mutants	# compiled	# executed	# mutants	# compiled	# executed	# mutants	# compiled	# executed
if(rtIsNaN(X)) → if(!rtIsNaN(X))	0	0	0	3	3	2	3	3	3
if(rtIsInf(X)) → if(!rtIsInf(X))	0	0	0	0	0	0	2	2	2
if(!rtIsNaN(X)) → if(rtIsNaN(X))	0	0	0	0	0	0	0	0	0
if(!rtIsInf(X)) → if(rtIsInf(X))	0	0	0	0	0	0	0	0	0
insert if statment - check divide by 0	0	0	0	0	0	0	18	16	16
remove if statment - check divide by 0	0	0	0	0	0	0	0	0	0
miultiply denominator by zero	24	24	24	24	24	24	31	31	31

Table 4.6: Mutation output details for Time and Space Model category in all three systems

Mutation Operator	Cruise control			Helicopter			Boeing 747		
	# mutants	# compiled	# executed	# mutants	# compiled	# executed	# mutants	# compiled	# executed
datatype of time is multiplied by 1000	6	5	5	18	17	17	20	17	17
datatype of time in ifstmt() is negated	4	4	3	4	4	3	0	0	0
variable of time is multiplied by 1000	2	2	2	2	2	2	3	3	3
variable of time in ifstmt() is negated	7	7	7	10	10	10	1	1	1
time and space variables	0	0	0	0	0	0	0	0	0

Table 4.7: Summary of the mutation output details for all three systems

Mutation Operator	Cruise control			Helicopter			Boeing 747		
	# mutants	# compiled	# executed	# mutants	# compiled	# executed	# mutants	# compiled	# executed
Total	283	279	277	416	408	406	840	822	820
Number of lines in file			279			449			1290
Number of unique mutated locations*			84			193			435
Total mutation coverage			30.10%			42.98%			33.72%
*Unique mutated location is the number of lines that got changed by the mutation tool.									

4.4 Results

We generated a total of 1539 mutations from the three different control models. Table 4.4, 4.5, 4.6, and 4.7 provides details on the number of mutants, the number compiled, and the number executed for each system. The tool covered a considerable percentage of the

Table 4.8: Summary of mutation results for all three systems

System name	Step input parameter name	Step input values	Most impacted step response characteristics	% Live mutants at 10% threshold	% Dead mutants at 10% threshold
Cruise control	Cruise speed (mph)	0-10	PeakTime	0	100
Helicopter	Pitch angle (deg)	0-1	SettlingMin	0	100
	Roll angle (deg)	0-1	SettlingMin	0	100
Boeing 747	Altitude (m)	61 - 62	RiseTime	95.37	4.63
	Airspeed (mps)	150.148 - 151.148	RiseTime	0	100

code, altering more than one-third of the code in each of the systems. Compilation errors were the result of rare syntax mismatches. Occasionally, a mutant would fail to execute due to a runtime error. For example, one of the errors was due to altering a timing value (the mutation changed a timing variable value to 0, as a result the program was waiting indefinitely) and the other was due to a change in random number generation (the mutation resulted in a code change seeding the random value to zero, causing the program to run in a loop indefinitely to find a valid number). Overall, 1503 mutants out of the 1539 mutations were successfully compiled and executed. This demonstrates the strength and robustness of our mutation tool.

4.4.1 Analysis I

To analyze the impact of our mutated code, we designed an oracle to classify the results as either "live" or "dead". To better understand how a normal mutation tool works, we refer the reader to [53]. Mutation testing is a check on the robustness of a software test suite. It does this by mutating software and comparing the output of mutated and original software. If changes are detected the test suite is robust, otherwise it should

be improved. Our **use** of mutation testing differs slightly as we assume an ideal test suite that will catch all errors, and are therefore interested in assessing the control code's robustness to small changes. As a result, 'live' mutants are desirable as they indicate there was no difference in outputs of mutated and original control code. 'Dead' mutants are undesirable as they indicate control code was vulnerable to small software changes.

The oracle has two criteria: one is the threshold value and the other is the number of step response quantitites. In our case, we use the 8 step response quantities from the system, we classify a system as "live" if all 8 step response quantities have an output value within a certain percent of the original design. If not, the mutant is considered "dead." We varied this threshold between 0% to 100% to capture the amount of variation in a step response that might be considered acceptable. Thresholds above 10% resulted in an inability to discriminate performance as all mutants would either be live or dead. Columns 5 and 6 in Table 4.8 show the percent live and dead mutants at the 10% threshold.

A summary of our findings are provided in Table 4.8, in the 2^{nd} column is the control input parameter, the step values of which are given in the 3^{rd} column. In the final column is the % of dead mutants. At just a 10% threshold output difference all mutants are killed by the test suite for all control systems except altitude control of the Boeing 747. **This shows the fragility of the control designs which generate different responses with even just a single, small change to the software**. In contrast, only 4.63% of mutants were killed in the altitude controller of the Boeing 747, suggesting that **this controller is very robust to small software changes**. Interestingly, only a few quantities in the system response were responsible for this dramatic change. For example, in the cruise control system, only the PeakTime quantity was not within the threshold limit as a result all the mutants were dead. Similarly, for the helicopter system, only the SettlingMin quantity was highly impacted by our mutations but caused all the mutants to die. Our investigation suggests that these two controllers are not robust to software changes and

the inevitable accompanying evolution. For the Boeing 747, on the other hand, airspeed was not affected by our mutations at all. Altitude was only mildly affected. This suggests a controller that is more robust to software changes and maintenance that are part of a healthy controller evolution. However, control system for airspeed of Boeing 747 was severely affected and RiseTime played a crucial role.

4.4.2 Analysis II

In this section, we present a more thorough analysis by designing a new oracle to compare our mutation output results. The analysis in section 4.4.1 is very conservative, in the sense, our previous oracle classifies a mutant as "dead" even if one of the step quantities does not satisfy the threshold criteria of 10% variation in results. Therefore, to better understand the controller performance, in this second analysis, we made two major changes in oracle. First, instead of 10% threshold, the oracle now uses a value from 0% to 100% in the increment of 10. Second, instead of having all 8 step quantities within the threshold value, the oracle now changes the number of step quantities within the threshold from 1 to 8 by gradually increasing the number by 1.

The results are presented in Figure 4.3, 4.4, 4.5, 4.6,and 4.7. The value present in each cell of these figures represents the percentage of mutants classified as either live or dead. On the x-axis, we have the number of step quantities within the threshold criteria. On the y-axis, we have the threshold criteria; it is given by the range of values from 0-100 denoting the % of error that the step response quantities are allowed to diverge. We begin our analysis by classifying a mutant as "live" even if one of the step quantities value is within the threshold. On the x-axis, we then continue our analysis by gradually increasing the number of step quantities within the threshold until all the 8 step quantities have to be within the threshold. Similarly, we repeat this process for different threshold values from 0% to 100%.

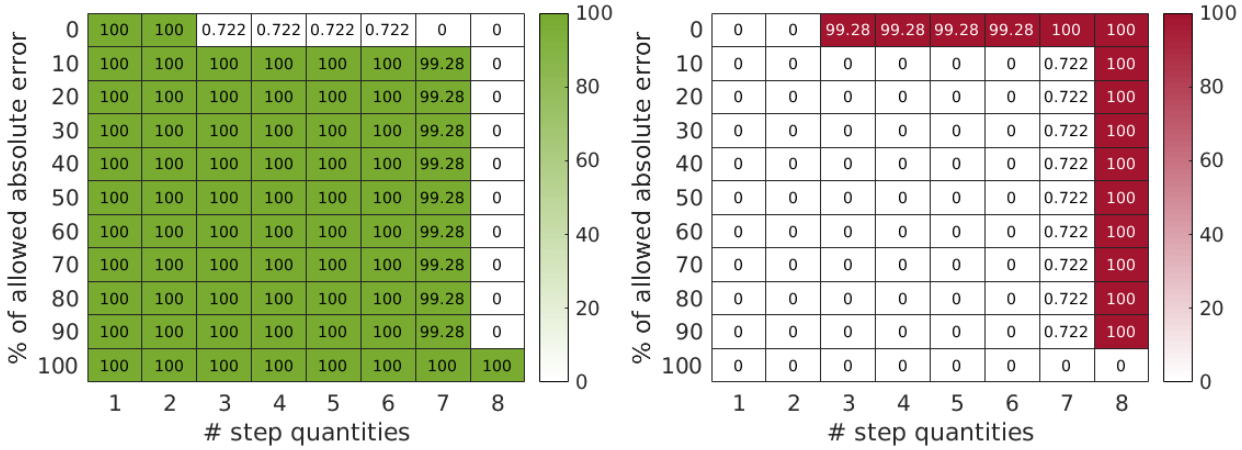
In addition to this, to show the impact severity on the controller performance, we have color coded each cells in the figures with green color and red color. Green color denotes that the mutants are alive, we have different shades of green color (lighter tone to darker tone) to show the level of impact severity, darker greener means that the system is more stable or less vulnerable to cyber-physical mismatches. For example, in Figure 4.7(a), altitude of Boeing 747 is the most robust system as it has more green color shades. On the other hand, the red color denotes that the mutants are dead, lighter red tone shows less impact whereas darker red shows critical impact in controller performance. For example, in Figure 4.6(b), airspeed of Boeing 747 is one of the most unstable system as it has more darker red color shades.

For the cruise control system, we found that the system was mostly stable and only one step quantity (i.e., PeakTime) played a major role in not meeting the criteria. The importance and criticality of this quantity vary depending on the application, but our analysis shows that a small variant of the system gets affected at a very early stage of 3 step quantity requirement, given by a value greater than zero in Figure 4.3(b). For helicopter system, both pitch angle and roll angle step response, most of the mutants got impacted at an early stage, given by a value greater than zero in Figure 4.4(b) and 4.5(b). Especially in the last 8 step quantity requirement, even though SettlingMin quantity had a negative value we did not consider this as ”live” because the mutant itself did not have a valid SettlingTime value. For Boeing 747 system, altitude step response was the most stable system we observed, as shown in Figure 4.7. On the other hand, the Boeing 747 airspeed quantity was the most unstable system we observed, denoted by more number of zeros and less values in % live mutants in Figure 4.6(a).

Overall, we observed that our mutation tool can expose the impact in controller performance. In other words, our generic templates is able to bring out the mismatches between the control design model and software system, causing the mutants to be unstable.

It is also interesting to note that if a system has more than one control parameter, like in the Boeing 747 system, it is not necessary that both parameter has to be impacted at a same level. This needs to be further investigated in the direction of dependency between two control parameters in a control system. In control engineering this analysis is called multiple-input-multiple-output (MIMO) system analysis. In summary, the analysis II supports the results from the analysis I, it also shows how small divergence can impact the controller performance with varying threshold criteria.

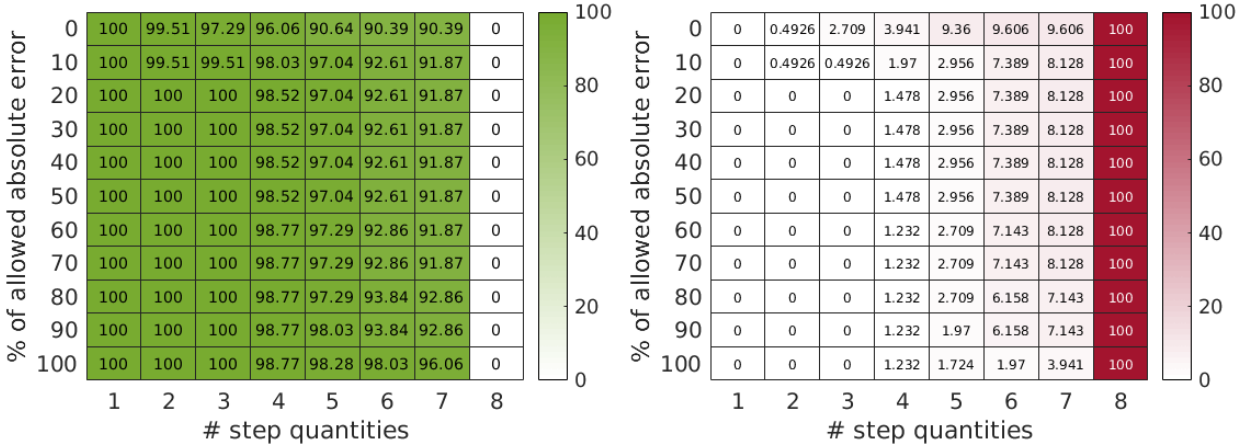
The key impact of our tool is that much like a change in control gain can be directly mapped to a change in system response [62], this tool allows us to directly map control software changes to a change in system response. This opens the door for studying how to design controllers that lead to robust software implementations.



(a) % Live mutants

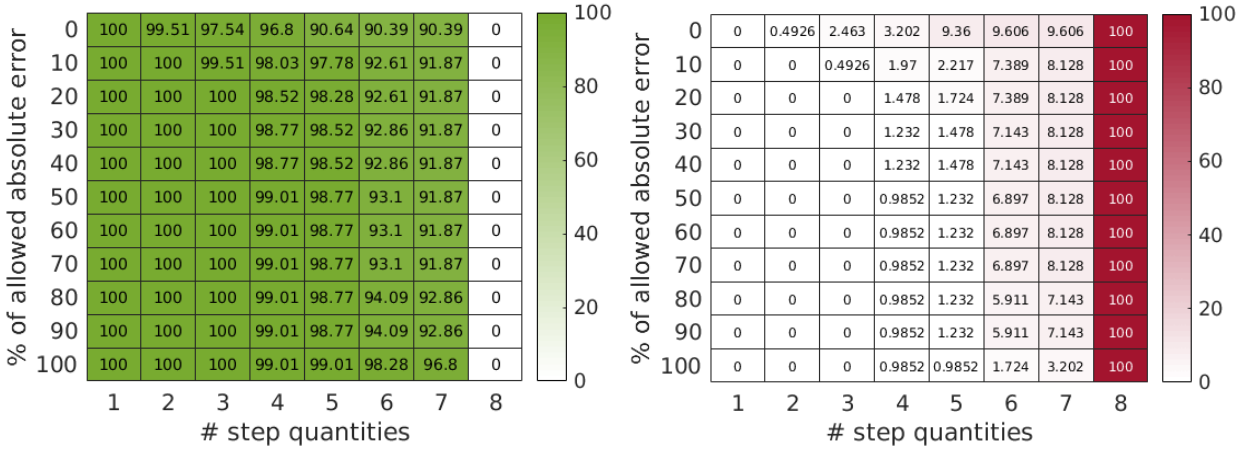
(b) % Dead mutants

Figure 4.3: Cruise control system divergence for speed step quantity



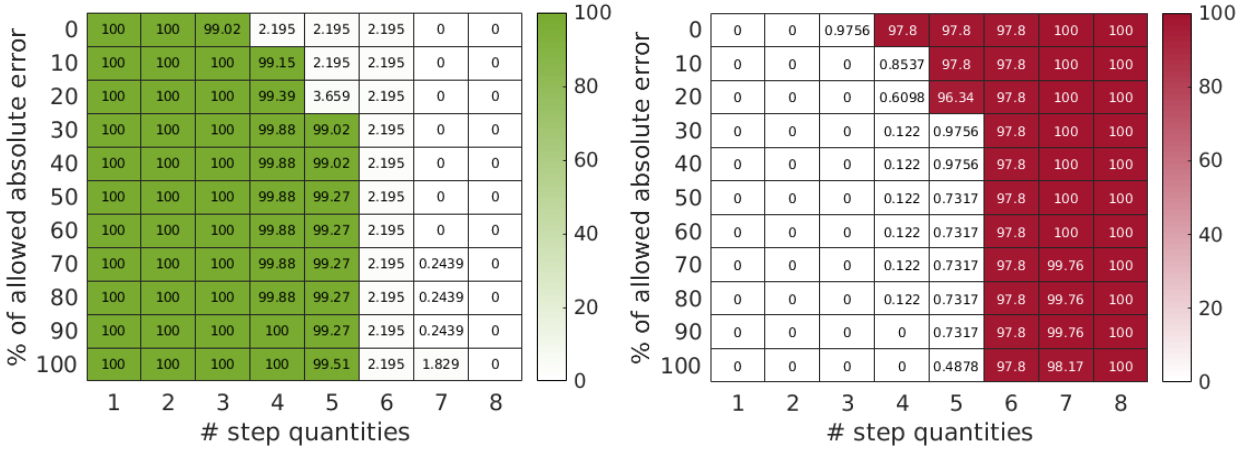
(a) % Live mutants (b) % Dead mutants

Figure 4.4: Helicopter system divergence for pitch angle step quantity



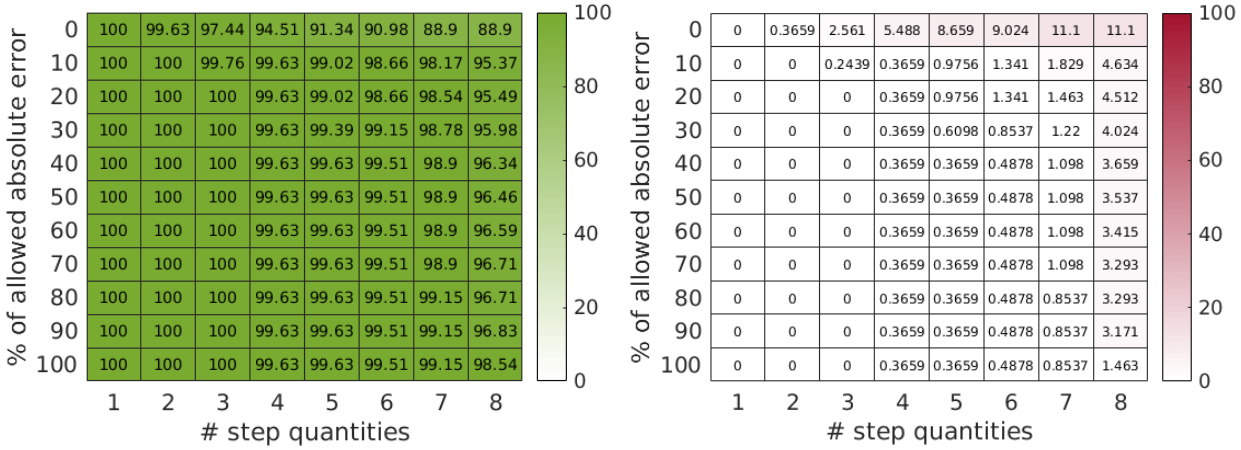
(a) % Live mutants (b) % Dead mutants

Figure 4.5: Helicopter system divergence for roll angle step quantity



(a) % Live mutants (b) % Dead mutants

Figure 4.6: Boeing747 system divergence for airspeed step quantity



(a) % Live mutants (b) % Dead mutants

Figure 4.7: Boeing747 system divergence for altitude step quantity

Chapter 5

Conclusions and Future Work

Modern control systems are complex integrations of computation and physical systems where software defines the relationship between them. As systems evolve, so do their controllers and control software though there is little understanding about how control software evolves. A deeper understanding of the *types* and *quantity* of evolution that occur in controllers can help the control and software communities develop new models and development strategies to maintain the integrity of key properties verified in the model and/or software.

We have directly studied the evolution, through 964 commits with 15,066 control software lines changed, of two dominant open-source control software suites, ArduPilot and Paparazzi UAV, used extensively in safety-critical unmanned autonomous systems. We found that control software evolves quickly, with controllers being entirely rewritten through their lifetime, and introduced categories capturing some of the inherent mismatches between typical control models and control software not previously identified. To facilitate more rapid study of this evolution we built a mutation tool that can rapidly change control code and compare its performance against the original designs. The impact of this tool is the ability to map software changes directly to controller performance, thereby paving the way for studying the design of controllers robust to software changes.

Currently, our approach and tool are still at an early development stage. We could

incorporate a richer set of templates, including those that attempt to describe resource attributes category. The tool could also be improved by adding support for different programming language paradigms. We are also interested in understanding the effect of mutation changes on the real-time system performance, for example, the scenario could include a drone flying in a predetermined trajectory. We will be exploring such improvement and further applying the tool to a larger number of systems.

In addition to this, we acknowledge that control engineers also write software coding for controllers, we see this trend even at the very early stage as students start taking courses in control engineering. Once the control engineering students develop control design model, they implement the model by writing software code in platforms such as MATLAB, Arduino, etc. To help these developers imminently, during the software development process, we are exploring the possibility to automatically warn these developers if they write codes related to cyber-physical mismatches. We strongly believe that this could be achieved by creating a plug-in (i.e., plug-in is a software component containing functions capable of adding specific features) that could be integrated inside the software developers programming environment to provide a real-time message warning system.

Lastly, from a control engineering perspective, one of the most crucial future directions is to extend the scope and application of our study. Currently, the scope of our study is limited to P, PI and PID controller, in the future, we want to include analysis from other types of controllers such as observer-based control, Fuzzy-logic, etc. From an application point of view, we also plan to understand the different properties of the controller. For example, knowing the transfer function of the controller can help us perform additional mathematical analysis to understand properties like the relationship between input and output, get a response of the system to any input, and know the poles and zeros for stability analysis. One possible way forward is to obtain a transfer function from software code using software abstraction techniques from software engineering

discipline. Techniques like theorem solver and numerical invariant detection could help us abstract the transfer function from software code into a mathematical equivalent.

Bibliography

[1] git Documentation. https://git-scm.com/doc. Accessed: 2017-09-30.

[2] ArduPilot. ArduPilot Open Source Autopilot, 2018. 1.2, 2.2, 3.1.1

[3] Karl Johan Åström and B Wittenmark. *Computer-controlled systems: theory and design*. Prentice-Hall New York, 1984.

[4] Richard Baker and Ibrahim Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, 2013. 2.2.2

[5] E Bini and G M Buttazzo. The optimal sampling pattern for linear control systems. *Automatic Control, IEEE Transactions on*, 59(1):7890, January 2014.

[6] Víctor Braberman, Nicolas D'Ippolito, Nir Piterman, Daniel Sykes, and Sebastian Uchitel. Controller synthesis: From modelling to enactment. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1347–1350, Piscataway, NJ, USA, 2013. IEEE Press. 2.2

[7] Justin M Bradley and Ella M Atkins. Coupled cyber-physical system modeling and coregulation of a cubesat. *IEEE Transactions on Robotics*, 31(2):443–456, April 2015.

[8] Giampiero Campa. Airlib, 2018. Accessed: 2018-10-11. 4.2

[9] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 332–343, New York, NY, USA, 2016. ACM. 2.1

[10] Darren Cofer and Steven Miller. DO-333 certification case studies. In *NASA Formal Methods Symposium*, pages 1–15. Springer, 2014. 2.1

[11] Jamal Daafouz, Pierre Riedinger, and Claude Iung. Stability analysis and control synthesis for switched systems: A switched Lyapunov function approach. *IEEE transactions on automatic control*, 47(11):1883–1887, 2002. 1

[12] Marco D'Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. *Analysing Software Repositories to Understand Software Evolution*, pages 37–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 2.2.1

[13] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 158–171. ACM, 1996. 2.2.2

[14] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[15] RTCA DO. 178b. 1992. *Software considerations in airborne systems and equipment certification. Radio Technical Commission for Aeronautics (RTCA)*, 1992. 2.2.2

[16] Emad Ebeid, Martin Skriver, and Jie Jin. A survey on open-source flight control platforms of unmanned aerial vehicle. In *Euromicro Symposium on Digital Systems Design*. IEEE, 2017.

[17] Sebastian G Elbaum and John C Munson. Code churn: A measure for estimating the impact of code change. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 24–31. IEEE, 1998.

[18] Tom Erkkinen and Bill Potter. Model-based design for DO-178B with qualified tools. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2009. 2.1, 2.2.1

[19] Eric Feron. From control systems to control software. *IEEE Control Systems Magazine*, 30(6):50–71, December 2010. 2.1, 2.2.1

[20] Eric Feron and Fernando Alegre. Control software analysis, part I open-loop properties. *arXiv preprint arXiv:0809.4812*, 2008.

[21] Eric Feron and Fernando Alegre. Control software analysis, part II: Closed-loop analysis. *arXiv preprint arXiv:0812.1986*, 2008.

[22] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 299–310, New York, NY, USA, 2014. ACM. 2.2

[23] G F Franklin, M L Workman, and D Powell. *Digital Control of Dynamic Systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998. 2

[24] Gregory A. Hall and John C. Munson. Software evolution: Code delta and code churn. *Journal of Systems and Software*, 54(2):111–118, 2000. 3.2.1

[25] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. TrueTime: Simulation of control loops under shared computer resources. *IFAC Proceedings Volumes*, 35(1):417–422, 2002. 3.2.2

[26] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. *HyTech: A model checker for hybrid systems*, pages 460–463. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. 2.2

[27] Heber Herencia-Zapana, Romain Jobredeaux, Sam Owre, Pierre-Loïc Garoche, Eric Feron, Gilberto Perez, and Pablo Ascariz. PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL. *NASA Formal Methods*, 7226:147–161, 2012.

[28] Bo Hu and Anthony N Michel. Stability analysis of digital feedback control systems with time-varying sampling periods. *Automatica*, 36(6):897–905, 2000.

[29] BX Huang and Furong WANG. Cyber physical systems: a survey. *Presentation Report, Jun*, 2008.

[30] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011. 2.2.2, 4.1, 4.1

[31] R. Jobredeaux, T. E. Wang, and E. M. Feron. Autocoding control software with proofs I: Annotation translation. In *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, pages 7C1–1–7C1–13, October 2011.

[32] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press. 2.1

[33] EI Jury and FJ Mullin. The analysis of sampled-data control systems with a periodically time-varying sampling rate. *Automatic Control, IRE Transactions on*, (1):15–21, 1959.

[34] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, Jan 2003.

[35] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003. 1, 2.1

[36] Bruce H Krogh. Cyber physical systems: the need for new models and design paradigms. *Presentation Report*, 2008.

[37] Marta Kwiatkowska, Gethin Norman, and David Parker. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice*, 15(11):1427 – 1434, 2007. Special Issue on Manufacturing Plant Control: Challenges and Issues. 2.2

[38] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997. 3.2.2

[39] Edward A. Lee. The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15(3):4837–4869, 2015. 3.2.2

[40] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press, 2016.

[41] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980. 2.2.1

[42] Richard J Lipton. Fault diagnosis of computer programs. *Student Report, Carnegie Mellon University*, 1971. 2.2.2

[43] Daniel Marjamäki. Cppcheck: a tool for static c/c++ code analysis, 2013. 4.3.2.1

[44] Johnny Marques and Adilson Marques da Cunha. Use of the RTCA DO-330 in aeronautical databases. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D1–1. IEEE, 2015.

[45] MathWorks. Helicopter System Documentation, 2018. Accessed: 2018-10-11. 4.2

[46] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Brcukmann. MiL testing of highly configurable continuous controllers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering - ASE '14*, pages 163–174, New York, New York, USA, September 2014. ACM Press.

[47] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, 57:705–722, January 2015. 2.2

[48] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. Effective test suites for mixed discrete-continuous stateflow controllers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 84–95, New York, New York, USA, August 2015. ACM Press.

[49] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Publishing Company, Incorporated, 1 edition, 2008. 2.2.1

[50] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005. 4.2

[51] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC counting standard. In *Cocomo Ii Forum*, volume 2007, pages 1–16, 2007. 2

[52] University of Michigan. Cruise Control System Documentation, 2018. Accessed: 2018-10-11. 4.2

[53] Jeff Offutt. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098–1107, 2011. 4.4.1

[54] Katsuhiko Ogata and Yanjuan Yang. *Modern control engineering*, volume 4. Prentice hall India, 2002. 4.3.2

[55] PaparazziUAV. PaparazziUAV, 2018. 1.2, 2.2, 3.1.1

[56] Libre Pilot. LibrePilot – Open – Collaborative – Free, 2018. 3.1.3

[57] pixhawk. Pixhawk Flight Controller Hardware Project, 2018. 2.2, 3.1.3

[58] Joseph Porter, Gábor Karsai, Péter Völgyesi, Harmon Nine, Peter Humke, Graham Hemingway, Ryan Thibodeaux, and János Sztipanovits. Towards Model-Based Integration of Tools and Techniques for Embedded Control System Design, Verification, and Implementation. In *MoDELS Workshops*, pages 20–34. Springer, 2008. 2.1

[59] Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013. 2.1

[60] Tariq Samad and Gary Balas. *Software-Enabled Control: Information Technology for Dynamical Systems*. John Wiley & Sons, 2003.

[61] Douglas C. Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006. 1

[62] Maria M Seron, Julio H Braslavsky, and Graham C Goodwin. *Fundamental limitations in filtering and control*. Springer Science & Business Media, 2012. 4.4.2

[63] Zhendong Sun and Shuzhi Sam Ge. Analysis and synthesis of switched linear control systems. *Automatica*, 41(2):181–195, 2005. 2.2.1

[64] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, April 1997.

[65] Jiafu Wan, Hehua Yan, Hui Suo, and Fang Li. Advances in cyber-physical systems research. *KSII Transactions on Internet and Information Systems (TIIS)*, 5(11):1891–1908, 2011.

[66] Fei-Yue Wang and Derong Liu. Networked control systems. *Theory and Applications, Springer-Verlag, London*, 2008.

[67] Björn Wittenmark, Karl Johan Åström, and Karl-Erik Årzén. Computer control: An overview. *IFAC Professional Brief*, 1, 2002. 2

[68] M. Zimmer, J. K. Hedrick, and E. A. Lee. Ramifications of software implementation and deployment: A case study on yaw moment controller design. In *2015 American Control Conference (ACC)*, pages 2014–2019, July 2015. 2, 2.1